Open in Colab

Programação Diferenciável (Aprendizado Profundo - UFMG)

Preâmbulo

O código abaixo consiste dos imports comuns. Além do mais, configuramos as imagens para ficar de um tamanho aceitável e criamos algumas funções auxiliares. No geral, você pode ignorar a próxima célula.

```
# -*- coding: utf8

import matplotlib.pyplot as plt
```

```
plt.rcParams['figure.figsize'] = (8, 5)

plt.rcParams['axes.axisbelow'] = True
plt.rcParams['axes.labelsize'] = 16
plt.rcParams['axes.linewidth'] = 1
plt.rcParams['axes.spines.bottom'] = True
plt.rcParams['axes.spines.left'] = True
plt.rcParams['axes.titlesize'] = 16
plt.rcParams['axes.ymargin'] = 0.1

plt.rcParams['font.family'] = 'serif'

plt.rcParams['axes.grid'] = True
plt.rcParams['grid.color'] = 'lightgrey'
plt.rcParams['grid.linewidth'] = .1

plt.rcParams['xtick.labelsize'] = 16
plt.rcParams['xtick.bottom'] = True
plt.rcParams['xtick.direction'] = 'out'
plt.rcParams['xtick.major.size'] = 10
plt.rcParams['xtick.major.width'] = 1
plt.rcParams['xtick.minor.size'] = 3
plt.rcParams['xtick.minor.width'] = .5
plt.rcParams['xtick.minor.visible'] = True

plt.rcParams['ytick.labelsize'] = 16
plt.rcParams['ytick.left'] = True
plt.rcParams['ytick.direction'] = 'out'
plt.rcParams['ytick.major.size'] = 10
plt.rcParams['ytick.major.width'] = 1
plt.rcParams['ytick.minor.size'] = 3
plt.rcParams['ytick.minor.width'] = .5
plt.rcParams['ytick.minor.visible'] = True

plt.rcParams['legend.fontsize'] = 16

plt.rcParams['lines.linewidth'] = 4
plt.rcParams['lines.markersize'] = 10
```

```
plt.style.use('tableau-colorblind10') # use um estilo colorblind!
plt.ion()
```

```
<contextlib.ExitStack at 0x7dc409922710>
```

1. Tensores em Numpy

O primeiro passo para usar numpy é importar a biblioteca.

```
import numpy as np
```

Quando pensamos no lado prático do aprendizado profundo, um aspecto chave que ajuda na implementação de novos algoritmos é a chamada programação diferenciável. Na próxima aula vamos voltar na mesma. No momento, o importante é salientar que a programação diferenciável faz uso extensivo de Tensores.

Um Tensor é uma generalização de matrizes para mais dimensões. Quando falamos de tensores, temos três casos especiais e um genérico que engloba os outros três:

1. **Escalar:** Um tensor de zero dimensões.

```
1 1
```

```
1
```

2. **Vetor:** Um tensor de uma dimensão.

```
1 np.array([1, 2])
```

```
array([1, 2])
```

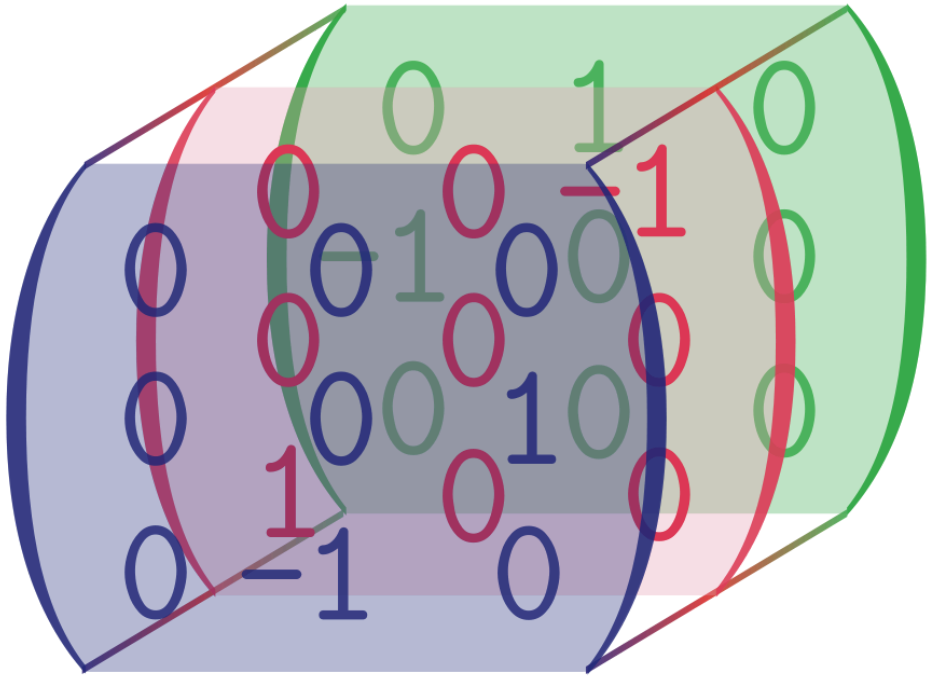
3. **Matrizes:** Um tensor de duas dimensões.

```
1 np.array([[1, 2], [3, 4]])
```

```
array([[1, 2],
       [3, 4]])
```

4. **Tensores**. Caso geral, representam n-dimensões. Na figura temos um tensor 3x3x3.

$$\epsilon_{ijk} =$$



No exemplo abaixo, temos um tensor de dimensão $3 \times 2 \times 2$.

```
1 X = np.random.randn(3, 2, 2) # Gera números aleatórios de uma normal N(0, 1)
2 X
```

```
array([[[ 0.40820746,  0.49601474],
        [-0.87209332,  0.65980131]],

       [[ 0.68900871,  0.44337452],
        [ 0.96842405, -0.52928973]],

       [[ 2.0153721 , -0.79644001],
        [-1.42448018,  0.6220849 ]]])
```

Note que ao selecionar elementos da primeira dimensão ficamos com matrizes 2×2.

```
X[0]
```

```
array([[ 0.40820746,  0.49601474],
       [-0.87209332,  0.65980131]])
```

```
X[1]
```

```
array([[ 0.68900871,  0.44337452],
       [ 0.96842405, -0.52928973]])
```

```
X[2]
```

```
array([[ 2.0153721 , -0.79644001],
       [-1.42448018,  0.6220849 ]])
```

1.1) Indexando

Sendo X uma matriz:

```
X = np.array([[1, 2], [3, 4]])
X
```

```
array([[1, 2],
       [3, 4]])
```

Podemos selecionar uma linha com a sintaxe `X[i]`, sendo `i` um inteiro.

```
X[0] # pegando a primeira linha de X
```

```
array([1, 2])
```

Podemos selecionar uma coluna com a sintaxe `X[:, j]`, sendo `j` um inteiro.

```
X[:, 1] # pegando a segunda coluna de X
```

```
array([2, 4])
```

Podemos selecionar mais de uma linha ou coluna utilizando a sintaxe `X[um_vetor]` ou `X[:, um_vetor]`, respectivamente.

```
X = np.array([[1, 2, 3], [4, 5, 6]])
X
```

```
array([[1, 2, 3],
       [4, 5, 6]])
```

```
cols = [0, 1]
X[:, cols] # iremos pegar a primeira e segunda colunas de X
```

```
array([[1, 2],
       [4, 5]])
```

Podemos selecionar linhas e colunas também indexando o tensor através de um vetor booleano.

A sintaxe `X[vetor_booleano]` retorna as linhas (ou colunas quando `X[:, vetor_booleano]`) onde o vetor é `True`.

```
X[[True, False]] # selecionamos apenas a primeira linha
```

```
array([[1, 2, 3]])
```

```
X[:, [True, False, True]] # selecionamos apenas a primeira e última coluna
```

```
array([[1, 3],
       [4, 6]])
```

1.2) Shape, Reshape e Ravel

Todo vetor, matriz e tensor pode ser redimensionado.

Observe como no tensor abaixo temos `3x2x2=12` elementos. Podemos redimensionar os mesmos para outros tensores com 12 elementos.

```
X = np.random.rand(3, 2, 2)
print(X.shape, X)
```

```
(3, 2, 2) [[[0.2827676  0.31646794]
  [0.15744129 0.60941124]]

 [[0.53342257 0.81578043]
  [0.14073367 0.88056941]]

 [[0.91197042 0.76521806]
  [0.56813487 0.71445641]]]
```

Podemos redimensionar os elementos como uma matriz 2×6:

```
X_matrix = X.reshape((2, 6))
print(X_matrix.shape, X_matrix)
```

```
(2, 6) [[0.2827676  0.31646794 0.15744129 0.60941124 0.53342257 0.81578043]
 [0.14073367 0.88056941 0.91197042 0.76521806 0.56813487 0.71445641]]
```

Em Numpy (e PyTorch), podemos fazer com que a biblioteca infira uma dimensão utilizando `-1`:

```
X_matrix = X.reshape((2, -1))
print(X_matrix.shape, X_matrix) # Note que também temos uma matriz de dimensão: 2x6
```

```
(2, 6) [[0.2827676  0.31646794 0.15744129 0.60941124 0.53342257 0.81578043]
 [0.14073367 0.88056941 0.91197042 0.76521806 0.56813487 0.71445641]]
```

Podemos redimensionar os elementos para um outro tensor:

```
X_tensor = X.reshape((6, 2, 1))
X_tensor.shape
```

```
(6, 2, 1)
```

E, por fim, podemos redimensionar os elementos como um vetor, realizando uma operação de `flattening`:

```
X_vetor = X.flatten()
X_vetor.shape
```

```
(12,)
```

Observação: Podemos redimensionar os elementos como um vetor de 3 formas principais:

1. Através da função `flatten`, como visto acima;
2. Através da função `ravel` (presente tanto em Numpy quanto em PyTorch);
3. Através de um `.reshape`, passando como parâmetro o número dos elementos (ou -1).

Os 3 métodos possuem algumas sutilezas que diferenciam um dos outros. Para mais informações consulte o seguinte link.

2. Tensores em PyTorch

PyTorch é o arcabouço que vamos usar para as nossas tarefas. Assim como o NumPy, o Pytorch é uma biblioteca de processamento vetorial/matricial/tensorial. Operações sobre os tensores do Pytorch possuem sintaxe consideravelmente parecida com operações sobre tensores do NumPy.

O mesmo faz uso de tensores bem similares ao NumPy. Porém, com PyTorch conseguimos fazer uso da GPU.

```
1 import torch
```

2.1) Casting para o dispositivo correto

Como usaremos processamento vetorial principalmente em GPUs para aprendizado profundo, primeiramente é possível verificar se há uma GPU disponível com o trecho de código abaixo, armazenando os tensores nos dispositivos apropriados.

```
1 if torch.cuda.is_available():
2     device = torch.device('cuda')
3 else:
4     device = torch.device('cpu')
5
6 print(device)
```

```
cpu
```

Podemos também realizar essa verificação em uma linha de código (+ pytônico)

```
1 device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
2 print(device)
```

```
cpu
```

2.2.) Tensores no Pytorch

Para criar tensores novos, podemos utilizar a função `torch.tensor`, similar à função `numpy.array` da biblioteca Numpy:

```
1 tns = torch.tensor([1, 2, 3, 4, 5, 6])
2 print(tns)
```

```
tensor([1, 2, 3, 4, 5, 6])
```

Podemos redimensionar os tensores de maneira similar ao que vimos em Numpy através da função `view`:

```
1 print(tns.view(2, 3))
```

```
tensor([[1, 2, 3],
        [4, 5, 6]])
```

Assim como mencionado em Numpy, podemos utilizar `-1` para fazer com que a biblioteca faça uma inferência no formato necessário de acordo com os elementos restantes:

```
# Note que obtemos o mesmo tensor anterior
print(tns.view(2, -1))
```

```
tensor([[1, 2, 3],
        [4, 5, 6]])
```

Observação: Em PyTorch, além de possuirmos a função `view`, também possuímos a função `reshape`. Ambas possuem algumas diferenças sutis. Caso queira obter mais informações sobre tais diferenças, acesse o seguinte link. Porém, no geral, iremos trabalhar mais com a função `view` ao desenvolver códigos utilizando PyTorch.

Podemos criar tensores previamente preenchidos por elementos, como 0s através da função `torch.zeros` e 1s através da função `torch.ones`:

```
tns_0 = torch.zeros(2, 3) # iniciando um tensor com 0s
tns_1 = torch.ones(2, 3) # iniciando um tensor com 1s

print(tns_0)
print(tns_1)
```

```
tensor([[0., 0., 0.],
        [0., 0., 0.]])
tensor([[1., 1., 1.],
        [1., 1., 1.]])
```

Similar à funções do Numpy, podemos iniciar tensores com valores aleatórios:

```
tns_u = torch.rand(2, 3) # valores que seguem uma distribuição uniforme no intervalo [0,1)
print(tns_u)

tns_n = torch.randn(2, 3) # valores que seguem uma distribuição normal N(0,1)
print(tns_n)

tns_perm = torch.randperm(6) # valores que são uma permutação aleatória no intervalo [0, 5]
print(tns_perm)
```

```
tensor([[0.2210, 0.0013, 0.7981],
        [0.3301, 0.3592, 0.4013]])
tensor([[-0.5441,  0.4866,  1.5819],
        [-1.9039,  0.4714, -1.6613]])
tensor([0, 1, 4, 2, 3, 5])
```

Similar à Numpy, podemos realizar operações de soma, multiplicação, entre outras, com tensores:

```
print(tns_u)
print(tns_n)

tns_sum = tns_u + tns_n
print(tns_sum)
```

```
tensor([[0.2210, 0.0013, 0.7981],
        [0.3301, 0.3592, 0.4013]])
tensor([[-0.5441,  0.4866,  1.5819],
        [-1.9039,  0.4714, -1.6613]])
tensor([[-0.3231,  0.4878,  2.3799],
        [-1.5738,  0.8306, -1.2600]])
```

Similar à Numpy, podemos indexar os tensores da mesma forma apresentada anteriormente:

```
print(tns_sum[1, 1]) # Indexando um elemento
print(tns_sum[0, :]) # Indexando uma linha (pode também ser tns_sum[0])
print(tns_sum[:, 1]) # Indexando uma coluna
```

```
tensor(0.8306)
tensor([-0.3231,  0.4878,  2.3799])
tensor([0.4878, 0.8306])
```

Podemos utilizar a função `torch.from_numpy` para converter um tensor de Numpy para PyTorch; ou a função `.numpy()` para converter um tensor de PyTorch para Numpy:

```
np_arr = np.random.randn(2, 3)
print(np_arr, np_arr.dtype)

torch_tns = torch.from_numpy(np_arr)
print(torch_tns)

arr = torch_tns.numpy()
print(arr)
```

```
[[ 1.18151116 -0.75561876 -1.03359314]
 [ 1.28988056  1.39383202 -0.36215711]] float64
tensor([[ 1.1815, -0.7556, -1.0336],
        [ 1.2899,  1.3938, -0.3622]], dtype=torch.float64)
[[ 1.18151116 -0.75561876 -1.03359314]
 [ 1.28988056  1.39383202 -0.36215711]]
```

Por fim, podemos concatenar dois, ou mais, tensores com a função `torch.cat`:

- O parâmetro `dim` em PyTorch é análogo ao parâmetro `axis` do Numpy. Nele, iremos informar sobre qual dimensão queremos que uma certa operação seja feita. No caso abaixo, ao informarmos a primeira dimensão (0), estamos dizendo para o PyTorch efetuar uma concatenação dos dois tensores a partir da sua primeira dimensão, ou seja, "colando" os tensores verticalmente (no sentido das linhas).

```
print((tns_0.shape, tns_1.shape))
tns_cat = torch.cat((tns_0, tns_1), dim=0)
print(tns_cat, tns_cat.shape)
```

```
(torch.Size([2, 3]), torch.Size([2, 3]))
tensor([[0., 0., 0.],
        [0., 0., 0.],
        [1., 1., 1.],
        [1., 1., 1.]]) torch.Size([4, 3])
```

Várias outras operações sobre tensores do Pytorch podem ser vistas nos seguintes tutoriais:

1. https://jhui.github.io/2018/02/09/PyTorch-Basic-operations/
2. https://pytorch.org/tutorials/beginner/pytorch_with_examples.html

Conjunto de Problemas 1: Vetorização

Antes de continuar, vamos importar algumas funções que serão utilizadas para testar o resultado dos seus algoritmos.

Estas, vão vir do módulo `testing` do numpy.

```
from numpy.testing import assert_equal
from numpy.testing import assert_almost_equal
from numpy.testing import assert_array_almost_equal
```

Seu objetivo é medir a velocidade das operações de álgebra linear para diferentes níveis de vetorização.

1. Construa duas matrizes A e B com entradas aleatórias Gaussianas de tamanho 128×256.

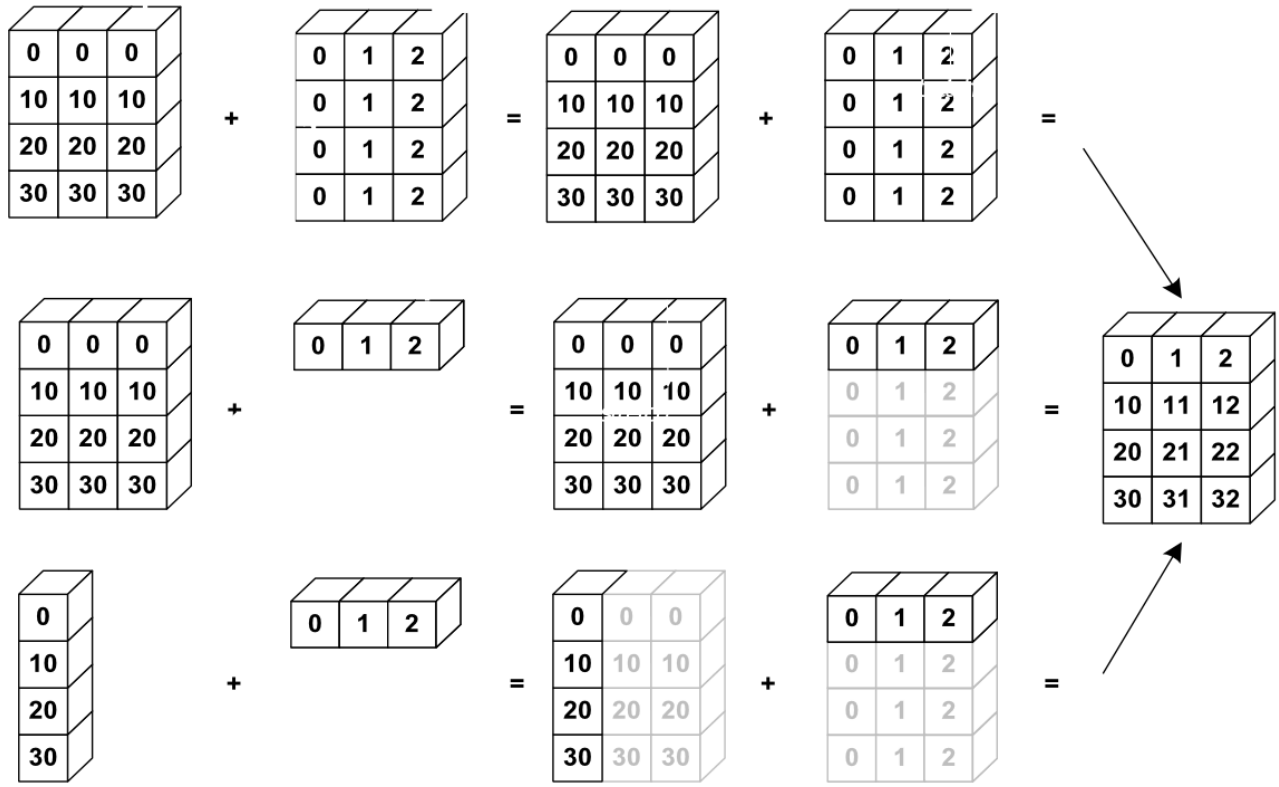
Dica: Use o módulo time para mensurar o tempo da operação.

```
A = torch.randn(128,256)
B = torch.randn(128,256)
```

```
# testes, não apague as linhas!!
assert_equal((128, 256), A.shape)
assert_equal((128, 256), B.shape)

# A chamada .numpy() converte os vetores em vetores numpy. Útil para testes!
Anp = A.numpy()
Bnp = A.numpy()

# testando média e desvio padrão
assert_almost_equal(Anp.mean(), 0, decimal=2)
assert_almost_equal(Anp.std(ddof=1), 1, decimal=2)

assert_almost_equal(Bnp.mean(), 0, decimal=2)
assert_almost_equal(Bnp.std(ddof=1), 1, decimal=2)
```

2. Calcule $C = AB^T$, tratando A como uma matriz, mas computando o resultado para cada coluna de B. Em outras palavras, realize um produto matricial utilizando um laço `for`! Pare realizar este código, é importante entender o conceito de broadcasting.

Em código numpy e torch, a operação de broadcasting replica linhas e colunas de tensores para realizar operações. Para entender melhor, leia o documento. A figura abaixo exemplifica broadcasting. No geral, as dimensões de arrays casam, as operações são realizadas (primeira linha da figura). Mesmo quando as dimensões não casem, se a última dimensão for compatível é feito a replicação (broadcasting), ver a segunda linha da figura. Por fim, mesmo quando as dimensões não casam mas uma delas é 1 (4x1 + 1x3 na linha 3), é feito broadcasting.



Dica: Você deverá fazer o código em uma linha apenas. Para isso, você vai focar no caso da linha 2 da figura. Multiplique uma linha de A por B. Depois disso, use `.sum(axis=...)` para realizar a soma na dimensão correta.

```
import time

start_time = time.time()
C = np.zeros((128, 128))
for linha in range(A.shape[0]):
    C[linha] = (A[linha]*B).sum(axis = 1)

print('Tempo de execução: ', time.time() - start_time)


```

```
Tempo de execução:  0.011831283569335938
```

```
# testes, não apague as linhas!!
Cteste = np.matmul(A, B.T)  # faz a leitura, realiza operação
assert_array_almost_equal(Cteste, C, decimal=3)
```

3. Calcule $C = AB^t$ usando operações matriciais. Ou seja, sem usar nenhum laço. Ao mensurar o tempo, ficou mais rápido?

```
start_time = time.time()
C = torch.mm(A, B.t())
print('Tempo de execução: ', time.time() - start_time)

```

```
Tempo de execução:  0.020906209945678711
```

Essa implementação é mais rápida.

```
# testes, não apague as linhas!!
Cteste = np.matmul(A, B.T)  # faz a leitura, realiza operação
assert_array_almost_equal(Cteste, C, decimal=3)
```

- Observando o tempo gasto nas duas formas de calcular $C = AB^T$, podemos perceber que ao utilizarmos funções nativas da biblioteca (como `np.matmul` e `torch.mm`), temos uma redução significativa do tempo se compararmos à utilização de um laço `for`.

Conjunto de Problemas 2: Computação eficiente de memória

Crie duas matrizes aleatórias de tamanho 4096×4096. Chame as mesmas de A e B novamente.

```
A = torch.rand(4096, 4096)
B = torch.rand(4096, 4096)
```

5. Crie uma função que recebe as matrizes A, B e C, e um número de iterações para atualizar C, de forma que $C = AB^T + C$. Essa função deve, primeiro, calcular a multiplicação de matrizes entre A e B e depois adicionar o valor à C, de acordo com o número de iterações. A mesma deve atualizar C sem alocar memória nova para essa variável.

```
def update_c(C, A, B, n_iter=2):
    D = torch.mm(A, B.t())
    for _ in range (n_iter):
        C[:] = D + C
```

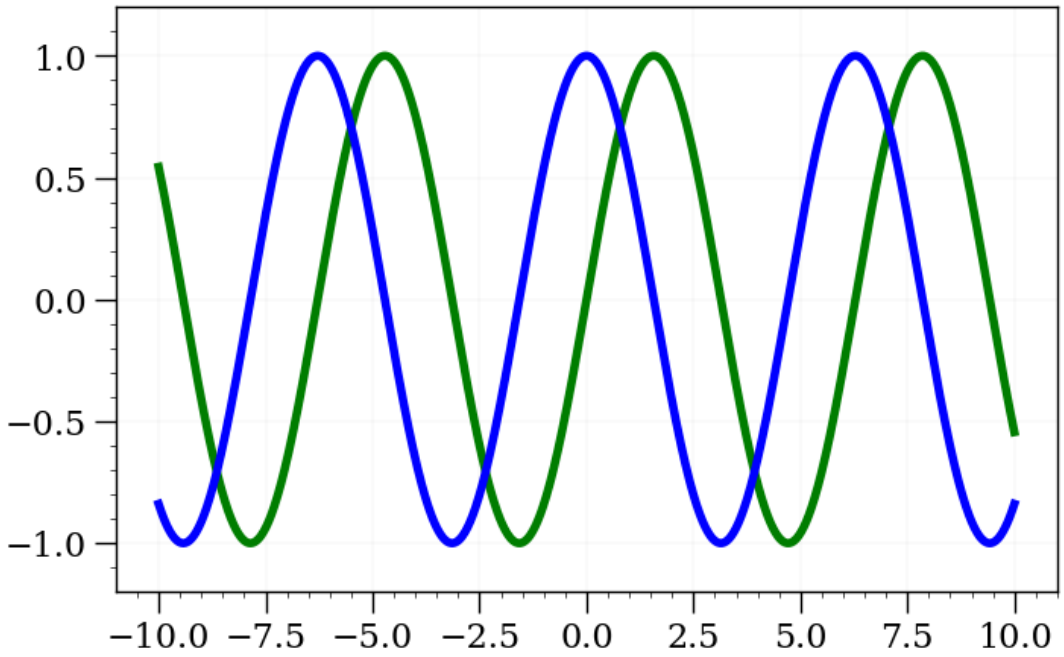
```
# testes não apague!
Ct = torch.zeros(A.shape)
Cteste = (Ct + np.matmul(A, B.T))
Cteste = (Cteste + np.matmul(A, B.T))

C = torch.zeros(A.shape)
update_c(C, A, B, 2)
assert_array_almost_equal(Cteste, C.numpy(), decimal=3)
```

Conjunto de Problemas 3: Programação Diferenciável

Agora vamos aprender um dos pontos chaves de fazer uso de bibliotecas como pytorch/tensorflow/etc, a programação diferenciável. Diferente do exercício que vocês fizeram na mão, usando a biblioteca conseguimos derivar de forma automágica. Portanto, observe como o código abaixo deriva a função seno.

```
x = np.linspace(-10, 10, 1000)
x_torch = torch.tensor(x, requires_grad=True)

y = torch.sin(x_torch)  # seno original

# Devido ao fato de que o y final não é um escalar precisamos passar o vetor v tal que v é a jacobiana
# pela qual vamos multiplicar as jacobianas da variavel node (neste caso, x) conforme descrito em
# https://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html
# veremos mais sobre o processo de backprogation, jacobiano, etc no futuro
v = torch.ones(x_torch.shape, dtype=torch.double)

# Como aqui x da origem a y diretamente, nosso v sera apenas um vetor de 1's com a mesma dimensao de x
y.backward(v)

plt.plot(x_torch.detach().numpy(), y.detach().numpy(), 'g', label='sin(x)')
plt.plot(x_torch.detach().numpy(), x_torch.grad.numpy(), 'b', label='sin\'(x)')
```

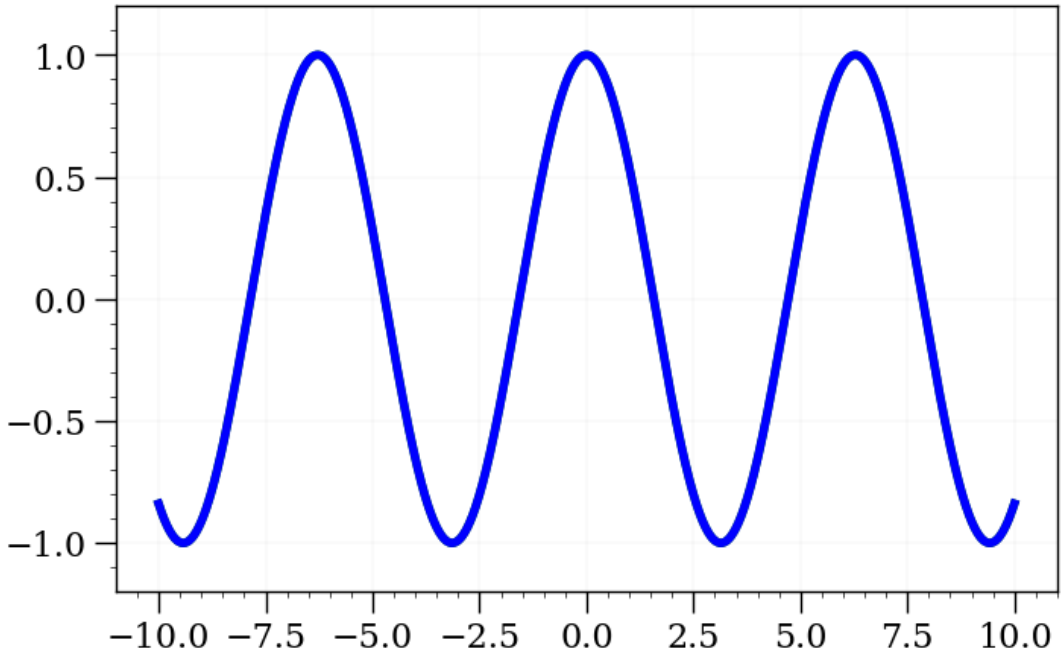
```
[<matplotlib.lines.Line2D at 0x7dc3e00e67d0>]
```



O resultado é a mesma curva da função cosseno! Para entender melhor o autograd, leia a seção respectiva do pyTorch Blitz.

```
plt.plot(x_torch.detach().numpy(), x_torch.grad.numpy(), 'g', label='sin\'(x)')
plt.plot(x_torch.detach().numpy(), torch.cos(x_torch).detach().numpy(), 'b', label='cos(x)')
```

```
[<matplotlib.lines.Line2D at 0x7dc33e981450>]
```



6. Derive a função logística usando pytorch.

$$f(x) = \frac{1}{1 + e^{-x}}$$

```
1 x = np.linspace(-10, 10, 1000) # Não mude o valor de x!
2 x_torch = torch.from_numpy(x)
```

```
1 x_torch = torch.tensor(x, requires_grad=True)
2 y = 1.0 / (1.0 + torch.exp(-x_torch))
3 v = torch.ones(x_torch.shape, dtype=torch.double)
4 y.backward(v)
```

```
1 # testes, não apagar
2 y_test = 1.0/(1 + np.exp(-x))
3 derivada_teste = y_test * (1 - y_test)
4 assert_array_almost_equal(derivada_teste, x_torch.grad.numpy(), decimal=3)
```

A operação *detach* permite quebrar a computação em várias partes. Em particular, isto é útil para aplicar a regra da cadeia. Suponha que $u = f(x)$ e $z = g(u)$, pela regra da cadeia, temos $\frac{dz}{dx} = \frac{dz}{du}\frac{du}{dx}$. Para calcular $\frac{dz}{du}$, podemos primeiro separar u da computação e, em seguida, chamar `z.backward()` para calcular o primeiro termo.

Observe no caso abaixo como derivamos $u = x^2$. A resposta deve ser $2x$ para cada termo `[0, 1, 2, 3]`.

```
1 x = torch.arange(4, dtype =torch.float)
2 x.requires_grad_(True)
3
4 u = x * x
5 jacobX = torch.ones(x.shape)
6 u.backward(jacobX)
7 x.grad
```

```
tensor([0., 2., 4., 6.])
```

Agora vamos fazer $z = u^3$ e computar as derivadas intermediarias.

```
x = torch.arange(4, dtype = torch.float)
x.requires_grad_(True)

u = x * x
v = u.detach()  # u ainda mantém o grafo computacional
v.requires_grad_(True)
z = v * v * v

print(z)

jacobX = torch.ones(x.shape)
u.backward(jacobX)
x.grad
```

```
tensor([  0.,   1.,  64., 729.], grad_fn=<MulBackward0>)
tensor([0., 2., 4., 6.])
```

Acima temos a derivada de x^2. Abaixo temos a derivada de $g(x^2)$.

```
jacobV = torch.ones(v.shape)
z.backward(jacobV)
v.grad
```

```
tensor([  0.,   3.,  48., 243.])
```

7. Agora, sendo $f(x) = 1 + x^2$ e $g(x) = 1 + 7f(x)^4$. Vamos aplicar a regra da cadeia em pytorch

```
x = torch.arange(4, dtype = torch.float)
x
```

```
tensor([0., 1., 2., 3.])
```

```
x.requires_grad_(True)
u = 1 + x*x
jacobX = torch.ones(x.shape)
u.backward(jacobX)
```

```
# Testando a sua solução
# Aqui pode ser meio confuso, mas o gradiente referente à derivada de
# f(x) com relação a x, por exemplo, fica armazenada no tensor x, mais especificamente em x.grad
assert_array_almost_equal([0, 2, 4, 6], x.grad.numpy())
```

```
v = 1 + x * x
u = v.detach()
u.requires_grad_(True)

z = 1 + 7 * torch.pow(u, 4)

jacobX = torch.ones(x.shape)
v.backward(jacobX)

jacobU = torch.ones(u.shape)
z.backward(jacobU)


```

```
# Testando a sua solução (caso seja necessário, modifique o nome da sua variável auxiliar para 'u')
# Note que agora o gradiente referente à derivada está presente na variável 'u', e não mais em 'x'
assert_array_almost_equal([28, 224, 3500, 28000], u.grad.numpy())
```

Conjunto de Problemas 4: Mais Derivadas

Vamos brincar um pouco de derivadas dentro de funções. Dado dois números x e y, implemente a função `log_exp`, que retorna:

$$f(x,y) = -\log\left(\frac{e^x}{e^x + e^y}\right)$$

```
def log_exp(x, y):
    numerador = torch.pow(torch.e, x)
    denominador = torch.pow(torch.e, x) + torch.pow(torch.e, y)
    return -torch.log(numerador/denominador)
```

1. Abaixo vamos testar o seu código com algumas entradas simples.

```
x, y = torch.tensor([2.0]), torch.tensor([3.0])
z = log_exp(x, y)
z
```

```
tensor([1.3133])
```

```
# Teste. Não apague
assert_almost_equal(1.31326175, z.numpy())
```

2. A função a seguir computa $\partial z/\partial x$ e $\partial z/\partial y$ usando `autograd`.

```
# O argumento funcao_forward é uma função python. Será a sua log_exp.
# A ideia aqui é deixar claro a ideia de forward e backward propagation, depois
# de avaliar a função chamamos backward e temos as derivadas.
def grad(funcao_forward, x, y):
    x.requires_grad_(True)
    y.requires_grad_(True)
    z = funcao_forward(x, y)
    z.backward()
    return x.grad, y.grad
```

Testando

```
x, y = torch.tensor([2.0], dtype = torch.double) ,torch.tensor([3.0], dtype = torch.double)
dx, dy = grad(log_exp, x, y)
```

```
assert_almost_equal(-0.7310586, dx.numpy())
assert_almost_equal(0.7310586, dy.numpy())
```

4. Agora teste com números maiores, algum problema?

```
x, y = torch.tensor([400.0]).double() ,torch.tensor([800.0]).double()
grad(log_exp, x, y)
```

```
(tensor([nan], dtype=torch.float64), tensor([nan], dtype=torch.float64))
```

5. Pense um pouco sobre o motivo do erro acima. Usando as propriedade de logaritmos, é possível fazer uma função mais estável. Abaixo segue a implementação da mesma. O problema aqui é que o exponencial "explode" quando x ou y são muito grandes. Este link pode ajudar.

```
x, y = torch.tensor([400.0], dtype = torch.double), torch.tensor([800.0], dtype = torch.double)
def stable_log_exp(x, y):
    return torch.log(1 + torch.exp(y-x))

dx, dy = grad(stable_log_exp, x, y)
```

```
1 stable_log_exp(x, y)
```

```
tensor([400.], dtype=torch.float64, grad_fn=<LogBackward0>)
```

```
1 # Teste. Não apague
2 assert_equal(-1, dx.numpy())
3 assert_equal(1, dy.numpy())
```

`